Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning

摘要

以往的研究提出了许多硬件预取技术，大多数都依赖于利用一种特定类型的程序上下文信息（例如，程序计数器，高速缓存行地址，或者高速缓存行地址之间的差异）来预测未来的内存访问。这些技术要么完全忽视预取器对整个系统（例如，内存带宽使用）的不良影响，要么将系统级反馈作为对系统无知的预取算法的后续考虑，由于先前的预取器无法在预取时考虑多种不同类型的程序上下文和系统级反馈信息，因此通常在广泛的工作负载和系统配置中丧失了其性能优势。在这篇文章中，他们提出了一个更全面的预取算法，该算法学习使用其设计固有的多种不同类型的程序上下文和系统级反馈信息进行预取。

基于这些考虑，论文提出了 Pythia [1]，将预取器形式化为一个强化学习的 Agent。对于每个需求请求，Pythia 观察多种不同类型的程序上下文信息来做出预取决策。对于每个预取决策，Pythia 接收一个奖励值，该奖励在当前内存带宽使用下评估预取质量。Pythia 使用这个奖励来强化程序上下文信息和预取决策之间的关联度，以便在未来生成高精度、及时且考虑了系统层次反馈的优质预取请求。

在对 Pythia 进行一系列的实验之后，与当前两种最先进的预取器（MLOP 和 Bingo）相比，Pythia 在单核中性能分别提高 3.4% 和 3.8%，在十二核的多核情况下分别提高 7.7% 和 9.6%，在带宽受限的配置中分别提高 16.9% 和 20.2%，同时只产生了 1.03% 的面积开销，并且在任何工作负载中都不需要软件进行干预。

关键词： 数据预取器；强化学习；SARSA

1 引言

处理器中缓存的预取器是为了匹配内存的访问 pattern，然后可以预先加载某些数据块到缓存中，这样后续的相关数据块的访问就能极大减少延迟，从而提高处理器性能，这是处理器中相当重要的一个部件。预取器为了快速的识别匹配这些访问 pattern，通常会使用一些程序内容相关的信息用于匹配一部分的内存请求，这些信息被叫做 feature。现有的很多工作通过利用各种各有的程序 feature，比如 PC、Address、Page Offset 等，用于提高预取器的覆盖率和正确率，及时且准确的预取可以减少内存的访问延迟从而提高处理器的整体性能。然而间歇性的预取请求可能会对整个系统造成一些不良的影响，比如增加了内存的访问带宽消耗、缓存污染、内存访问干涉等，这些反而会降低，所以一个好的预取器要在获得较好的性能的同时降低这些副作用的出现。

绝大多数的预取器都有这样的三个不足之处：

- 大多数只使用单一的 program feature 来进行预测：这样的做法会导致预取器在某些 workload 上具有较好的性能，而在其他不相关的 workload 上反而体现不出来。
- 缺少固有的系统意识（没有考虑系统的带宽使用等）：在内存带宽受限的情况下，激进的预取策略反而会因为内存带宽的限制，使得整体缓存的访问延迟增加，性能不增反降。
- 缺少能够自适应多种不同 workload 以及系统配置的能力：不同的 workload 具有不同的 feature 特征，在先前的一些启发式的预取器中往往只能利用一种程序特征而不能动态调整，对于不同的 workload 体现出不同的兼容性。

而关于上面的这些不足之处，正是本文 Pythia 将会解决的地方。接下来我们讨论强化学习在预取中是如何发挥其作用的。

首先我们知道，强化学习（Reinforcement Learning，RL）是一种在给定环境中快速学习如何获得最大奖励的机器学习算法，强化学习的目标就是在一定时间范围内获得最大累计奖励，在不断地与环境的交互中，得到反馈，调整算法参数，最终得到良好的训练效果。

预取器的表现好坏不止需要看 coverage 和 accuracy，还要看对系统所造成了负面影响，比如内存带宽的占用，一个预取器仅仅能达到高的 accuracy 是不充分的，这并不能完全代表一个预取器的好坏，预取器应该是 performance-driven 的，且能够在 coverage 和 accuracy 中动态进行 trade-off，在复杂状态空间中预取的自适应和性能驱动性质，使得 RL 能够适用，并且自动在与系统的交互中学习得到最佳的策略。RL 的 agent 并不需要进行 offline training，是一种 online learning 的策略，预取器也需要能够根据 workload 动态调整相关策略，同时又因为强化学习是一种较为容易在硬件中部署的机器学习算法，这一系列因素使得 RL 在预取上面能够有极大的发挥空间。

2 相关工作

在缓存预取这个研究方向上，有着各种基于不同原理的预取器存在，对于传统的预取器，可以分 precomputation-based、基于 temporal 时间、基于 spatial 空间这几种形式的预取器，下面针对这三种形式进行概述：

- precomputation-based
  利用提前执行程序代码来生成未来的内存请求。这类预取器包括 runahead execution 和 helper thread execution，这类预取器的主要优点是可以在程序内存访问模式没有可识别规律时，仍能生成高度精确的预取，不依赖程序的内存访问模式。但是这类预取器也存在一些缺点，如设计复杂度高，需要额外的处理资源来实现提前执行，实现难度大，调优复杂等

- temporal 时间
  记录一段较长的 cache address 访问序列，这时候如果一个前面记录的 address 再次出现了，temporal 预取器就会查询前面遇到这一地址之后的访问地址，然后将这个地址作为预取地址。缺点是这种预取器由较高的存储需求

- spatial 空间
  这种类型的预取器主要通过预测 cacheline 的 delta 或者 spatial bit-pattern 来实现的，需要学习程序运行时候不同内存区域的 access pattern。spatial 预取器提供了较高的 accuracy，具有更小的存储要求

在机器学习于体系结合这一层面上（Machine Learning(ML) in Computer Architecture），也有着许多的相关应用，如可以用于设计自适应的数据驱动算法，有研究人员提出使用机器学习技术设计各种微体系结构任务的算法, 如内存调度、缓存管理、分支预测等，这些工作展示了机器学习在提高硬件系统自适应性方面的潜力。在探索大规模的微体系结构设计空间上，也有着广泛的应用，研究者探索使用机器学习技术自动搜索最佳的微体系结构设计, 如 NoC 结构优化、芯片放置优化等。这可以大幅减少设计空间探索所需的工程时间。相比之下,Pythia 使用了强化学习方法进行硬件预取, 具有以下优点:

- 可以同时学习和使用多种程序特征与系统反馈
- 可以在线自定义配置
- 硬件实现开销低

3 本文方法

论文中提出的基于强化学习的缓存预取器 Pythia 有如下的一些创新点：

- 可以整体考虑使用多种不同类型的程序 feature，同时也会使用 system level 的反馈信息，这篇文章中作者使用了一个这样的信息：memory bandwidth usage
- Pythia 将硬件预取转化为了一个强化学习（Reinforcement Learning，RL）问题，Pythia 将会通过与处理器以及内存子系统的交互自动学习如何进行预取，Pythia 每次做出决策（进行或者不进行预取），就会得到一个 reward，用于评判它做出的行为的好坏以及是否及时（timeliness of the prefetch action），以此来动态适应各种不同程序的特征
- 可以在不修改 RTL 的基础上通过简单的配置寄存器自定义不同程序 feature 或者更改预取器的目标，只需要指定哪些 feature 可能会被使用到，具体的可以在指定学习目标（需要达到什么性能）后交由强化学习进行选择，不需要花太多时间在决定使用什么 feature 上面

3.1 Pythia 的工作原理

首先对于每一个 timestep 都可以对应于 Pythia 看到的一个新的 demand request，对于每一个 demand request Pythia 需要通过观察处理器的状态以及 memory subsystem 的情况来决定预取的动作，Pythia 所做出的每一个动作，都会获得一个 numerical reward（经过 accuracy 和 timeliness 评估后得到的），Pythia 的目标是为了寻找到最佳的预取策略从而最大化 accurate 和 timely 的预取请求，同时也会考虑 system-level feedback 相关信息，我们可以从下图看到这样一个大致的框架。

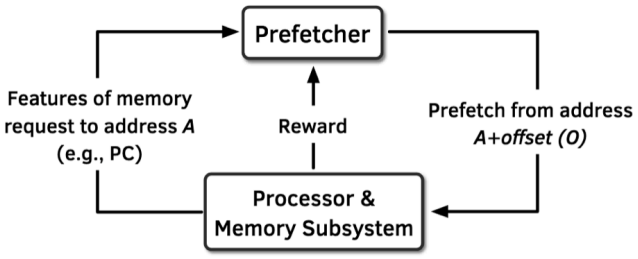

图 1. Pythia 架构

基于强化学习的预取器的三个支柱：State、Space、Action。对于 State，Pythia 中可以用一个 k 维的程序 feature，以一个 vector 的方式表示：

$$S \equiv \phi_S^1,\ \phi_S^2,\ ...,\ \phi_S^k$$

这些 feature 可以从两个主要的部件中取得，一个是 control-flow 部件（与程序的控制流有关，如 load-PC、branch-PC），一个是 data-flow 部件（与数据相关，如 cache line address、physical page number）。对于 Action，Pythia 中 RL-agent 的 Action 被定义为所选择的 prefetch offset（delta），使用 offset 作为 Action 可以极大减少 action space 的大小，其中 offset 为 0 就代表不进行预取。对于 Reward，这个是强化学习算法的一个核心的地方，Pythia 中可以按照下面的方式进行分类：

- Accurate and timely R_{AT}
  预取的请求在预取回填后被使用了
- Accurate but late R_{AL}
  预取的请求在预取回填之前就被使用了
- Loss of coverage R_{CL}
  预取的请求和 demand access 在不同的 physical page 中
- Inaccurate R_{IN}
  预取的块在一个时间窗口内并没有被 demand 两个 sub-level：
  - inaccurate given low bandwidth usage R_{IN}^L
  - inaccurate given high bandwidth usage R_{IN}^H
- No-prefetch R_{NP}
  Pythia 决定不预取的时候会得到这个 reward 两个 sub-level：
  - no-prefetch given low bandwidth usage R_{NP}^L
  - no-prefetch given high bandwith usage R_{NP}^H

对于上面这些 Reward，R_{AT} 和 R_{AL} 用于指导 Pythia 生成更加准确且及时的预取请求，R_{CL} 用于指导 Pythia 生成的预取请求位于 physical page 之内，R_{IN} 和 R_{NP} 可以提供相对应的 memory bandwidth usage feedback。

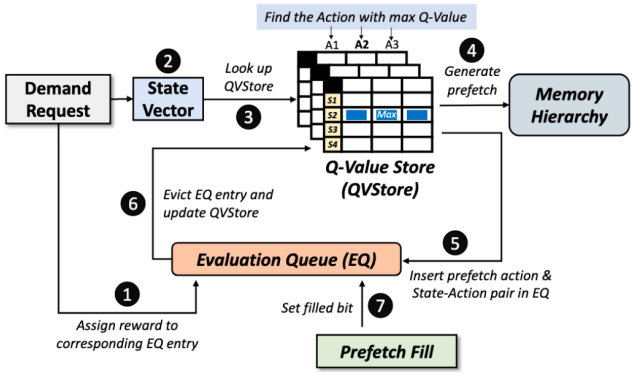

图 2. Pythia 工作流程

3.2 Pythia 的工作流程

Pythia 的具体结构包括两个主要硬件结构，一个是 Q-Value Store（QVStore），这个主要是记录了所有的 state-action 对的 Q-value，第二个是 Evaluation Queue（EQ），是一个 FIFO，记录了 Pythia 最近采用的 action，每一个 EQ entry 保存了三个主要的信息：（1）采用的 action；（2）对应 action 生成的预取地址；（3）the filled bit，用于表示这个预取请求是否已经被回填到 Cache 中。

如图 2 所示，Pythia 的工作可以分为七个主要的步骤：

1. 首先用 demand memory address 来需要检查 EQ 中的 enrty
   如果 EQ 命中了（地址存在），则表示之前已经在这个地址上发送了预取请求，这就表示 Pythia 生成了一个有效的预取，同时在这个时候 Pythia 会给对应 EQ Entry 上的 R_{AT} 或者 R_{AL} 赋值（具体取决于 filled bit）

2. 从 demand request 中提取 state-vector（状态的提取）

3. 根据 state-vector 寻找 QVStore 中对应的最高 Q-value 的 action

4. 根据这个 action 发送预取请求到 cache 中

5. 寻找最高 Q-value 的同时会将被选择的 prefetch action、prefetch memory address 和 state-vector 插入到 EQ 中
   需要注意的是对于那些 no-prefetch action 和那些在当前 physical page 之外的也会被插入到 EQ 中，同时这些 action 的对应 reward 也会被分配到 EQ 中

6. 当 EQ entry 被 evict 的时候，EQ 中存储的 state-action 对和 reward 会被用于更新 QVStore

7. 当预取请求得到回填的时候，Pythia 会利用 prefetch address 来检查 EQ 中 entry，然后 set 对应的 filled bit
   这个 filled bit 会在（1）中被用到，用来判断这个预取是 timely 还是 late 的，Pythia 需要不断追踪最近的 action 的状态，因为这些 action 并不是马上就能得到 reward

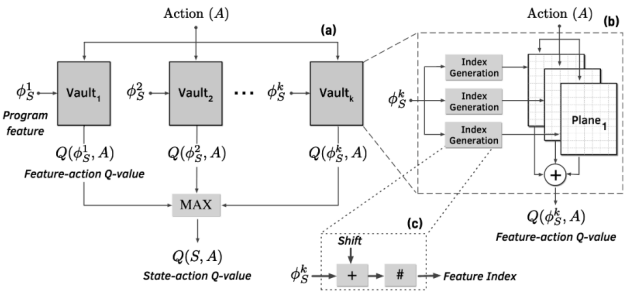

图 3. Q 值获取

3.3 Pythia 的具体设计

Pythia 为了能够在硬件上进行实际的部署，针对其中的 QVStore 进行了许多硬件上的考量，文章提出了一种基于 table based、hierarchical 的 state action paris 的存储结构（QVStore），QVStore 由多个 vault 组成，每个 vault 对应 state vector 中的一个特征，用于记录该特征下所有 action 的 Q 值（一个 state vector 中有多个 feature），在获取 Q-value 的时候每一个 vault 都是并行访问的。查询一个 state-action 对的 Q 值时，从每个相关的 vault 中查找该动作的特征 Q 值，然后取 max 作为最终的 state-action Q 值，为了缩减 vault 的大小，每个 vault 再分为多个 plane。每个 plane 仅存储 feature-action 对的部分 Q 值，这样的分层组织方式可以高效支持更高维的 state 空间，只需添加更多的 vault 即可。使用 vault 来存储所有的 feature 会随着 feature size 的增大而指数增大，这使得 feature 一增多就会导致最后的存储 table 难以部署到具体硬件上，可以使用 tile coding 的方式来解决这个问题，将 feature space 组织成多个小 tile，但是这需要再 feature 值的精度上做妥协。

Pythia 中参考 tile coding 的思想将一个 vault 组织 N 个二维的 table，每一个 table 称为一个 plane，每个 plane 的 entry 存储了一个 feature action pair 的部分 Q-value（上面提到的一个个 tiling），为了取回 feature-action Q-value，首先需要对 feature 进行 shift（这个 shift 的值是在设计的时候随机取的），然后进行 hash 后得到最终的 feature index，与 action index 一起组合起来可以用于取回 Q-value，最后的 Q-value 可以用将所有 plane 中的 partial Q-value 进行求和后得到。

QVStore 的搜索上，Pythia 有这几个步骤：(1) 使用当前 demand requst 的 state-vector 搜索 QVstore；(2) 搜索具有最大的 state-action Q-value 的 action（寻找最大值）。搜索 QVStore 是 Pythia 的关键路径，这也直接影响了 Pythia 的预取延迟，所以对着一部分逻辑进行流水线处理是很有必要的，对此作者对其进行了下面的流水线处理，将 Pythia 分为了 4 个流水线 Stage：

1. Pythia 还维护了一个 Q_{max} 来记录当前获得的最大的 Q-value，每次取回 Q-value 的时候会与其进行对比
2. Stage 0：利用给定 state-vector 的 feature 来计算 index
3. Stage 1：利用 feature indices 和 action index（要来索引二维数组）从每个 plane 中取回 partial Q-value

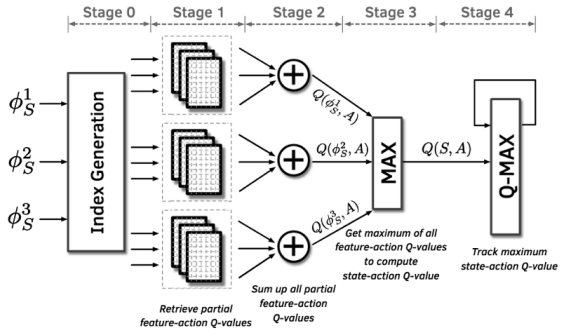

图 4. Q 值获取流水线

4. Stage 2：将所有从 plane 中取回的 partial Q-value 相加起来得到 feature-action Q-value 这是整个流水线中逻辑最多的地方，也是整个流水线的瓶颈

5. Stage 3：从所有的 feature-action Q-value 中计算最大值来得到 state-action Q-value

6. Stage 4：将 Stage 3 中获得的 Q-value 和 Q_{max} 进行对比，进而更新 Q_{max}

最后就是在具体 feature 的选择上所做的工作，feature 的选择也是一个重要的工作，作者将多种可能有贡献的程序 feature 提取出来，两两为一组作为 Pythia 的 feature 分别进行各种 workload 的测试，选取了其中性能表现最好的几个作为最后 Pythia 的参数选择，在 Pythia 中最后使用了 PC+Delta 和 Sequence of last-4 deltas 这两个 feature，用户也可以根据需要自行调整，这些在 Pythia 中都是保留了最大的自由度的。

4 复现细节

4.1 与已有开源代码对比

在他们所进行的实验中，只评估了在固定 LLC（Last Level Cache）缓存替换策略（Replacement Policy）下的各个预取器性能情况，没有评估不同 LLC 缓存替换策略中的性能情况，而替换策略在不同的处理器中有不一样的实现方式，所以这里将会在原有论文的基础上，增加不同替换策略在 Pythia 预取器中的表现情况，这样能够更全面地评估论文所提出的预取器的性能表现。

4.2 实验环境搭建

使用 ChampSim 模拟器复现了 Pythia 预取器，并实现了 Bingo、MLOP、SPP、Pythia 等预取算法，运行 SPEC CPU2006/2017 的 trace，同时作为修改，原有的实验中在 LLC 上使用的是 SHIP 替换算法，这里对 LLC 实现了常见的 LRU 算法以及 DRRIP 算法，所以一共需要做六组实验（单核-LLC-SHIP、单核-LLC-LRU、单核-LLC-DRRIP、多核-LLC-SHIP、多核-LLC-LRU、多核-LLC-DRRIP），实验平台硬件环境如下表所示：

ChampSim 是一款高度可配置的模拟器，可以配置各种处理器的体系结构相关参数，不同参数下 ChampSim 模拟器的运行行为也会有所不同，ChampSim 模拟的是 OoO（Out of

表 1. 实验硬件平台配置

操作系统	Ubuntu 20.04.6 LTS
CPU	AMD EPYC 7773X 64-Core
内存	252 GB
gcc 版本	9.4.0

Order，乱序）处理器，所以可配置的参数较为丰富，在本实验中，ChampSim 的配置参数如表 1 所示。

ChampSim 是一款高度可配置的模拟器，可以配置各种处理器的体系结构相关参数，不同参数下 ChampSim 模拟器的运行行为也会有所不同，ChampSim 模拟的是 OoO（Out of Order，乱序）处理器，所以可配置的参数较为丰富，在本实验中，ChampSim 的配置参数表 2 所示。

表 2. Champsim 配置

Core	1/4 Core, 4-wide OoO, 256-entry ROB, 72/56-entry LQ/SQ
BranchPerd	Perceptron-based, 20-cycle misprediction penalty
L1/L2 Caches	Private, 32KB/256KB, 64B line, 8 way, LRU, 16/32 MSHRs, 4-cycle/14-cycle roud-trip latency
LLC	2MB/core, 64B line, 16 way, SHiP/LRU/DRRIP, 64 MSHRs per LLC Bank, 32-cycle round-trip latency

5 实验结果分析

评估处理器性能的指标为 IPC（Instructions Per Cycles），IPC 越高处理器的性能也就越高，在控制其他不变量的情况下，只改变预取算法，那么最后 IPC 反映的就是预取器的性能高低。

图 5、6 中是单核情况下的性能情况，可以看到 Pythia 的性能对比四个预取器大部分情况下性能都是更优的，在一些 workload 中性能更有显著的提高，这说明 Pythia 的这种基于强化学习的预取策略能够有效适应不同 workload 的特性，带来稳定的性能提升，同时也可以看到其他的预取器性能表现在各个 workload 中各有差异，因为这些预取器都是基于启发式的策略，只能适应某些特定的 workload 从而带来性能的提升。

图 5. Pythia 单核 IPC 数据（ship 替换算法）

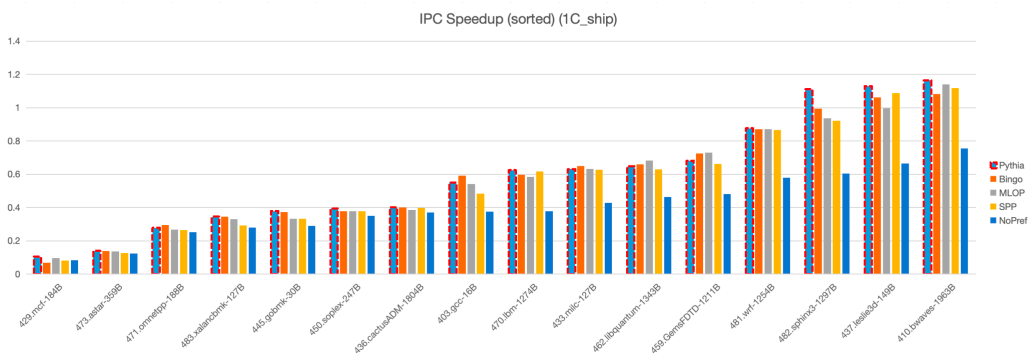

图 6. Pythia 单核 IPC 数据（ship 替换算法）（按照 IPC 排列）

图 7、8 是四核情况下的性能情况，多核情况下缓存的负载会更大，这也是最能反映预取器综合性能的测试，因为有的预取器会强调单核性能，在多核上的性能较差，一个优秀的预取器应该在单核和多核上都能有良好的性能表现，从这两个图中可以看到 Pythia 仍旧有着较强的性能表现，在一些用例中甚至能比 SPP 预取器领先数十个百分比，多核情况下 Pythia 仍然能够学习处理器运行时候的 workload 特性，带来稳定的性能提升。

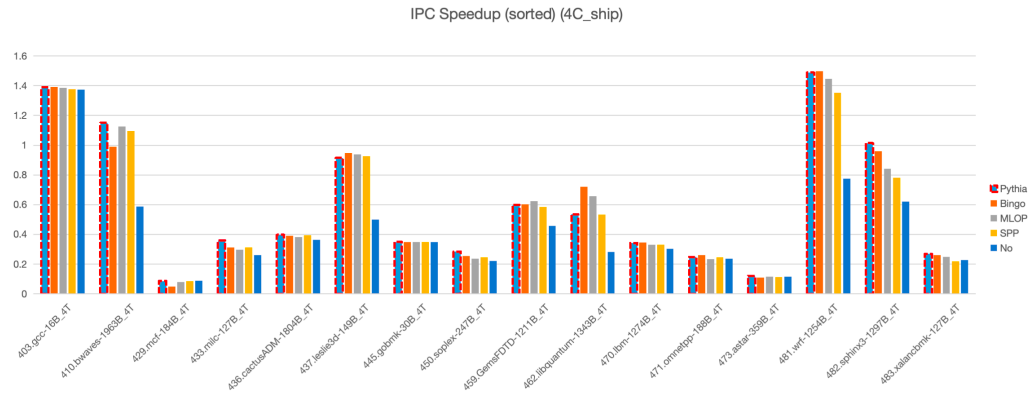

图 7. Pythia 四核 IPC 数据（ship 替换算法）

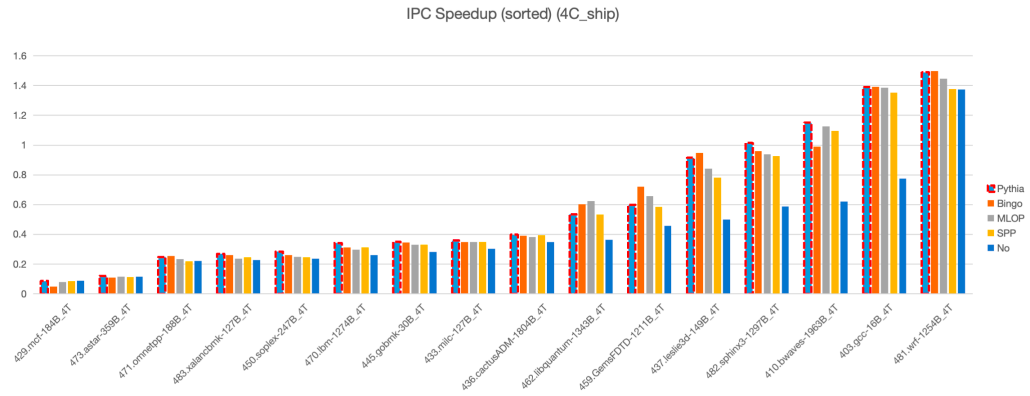

图 8. Pythia 四核 IPC 数据（ship 替换算法）（按照 IPC 排列）

下面我们将考虑改变 LLC 的替换策略对不同预取器的性能表现情况，由图 9、10 可以看到单核情况下 SPP 预取器对 LLC 的替换策略没有显著的反馈，不管是使用保守的 LRU 还是先进的 SHIP、DRRIP 都没有明显的性能影响，也不会带来显著的性能提升，而 Pythia 则能够适应不同 LLC 替换策略带来的增益，在 LLC 为 SHIP 替换策略下的表现最好，DRRIP 次之，说明 Pythia 能够从考虑多种因素适应不同的程序和整体系统的特性。

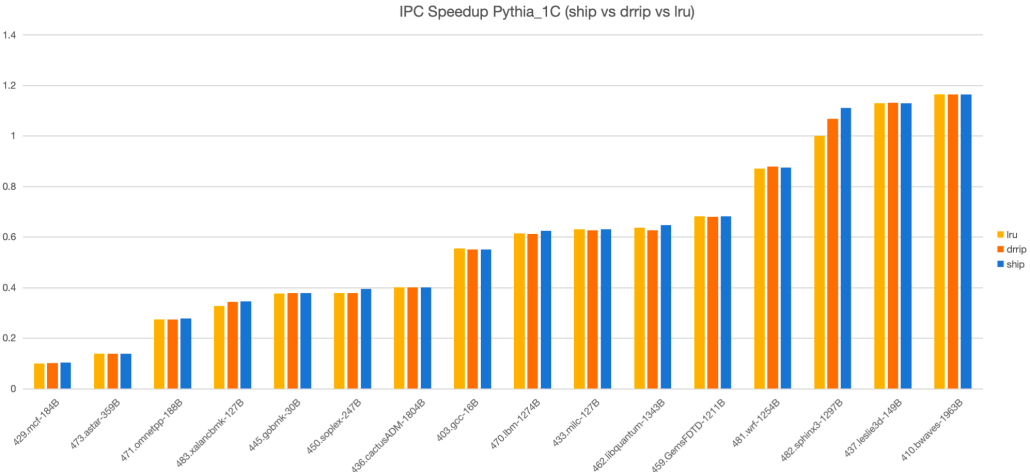

图 9. 单核不同替换算法下 Pythia 的 IPC 数据

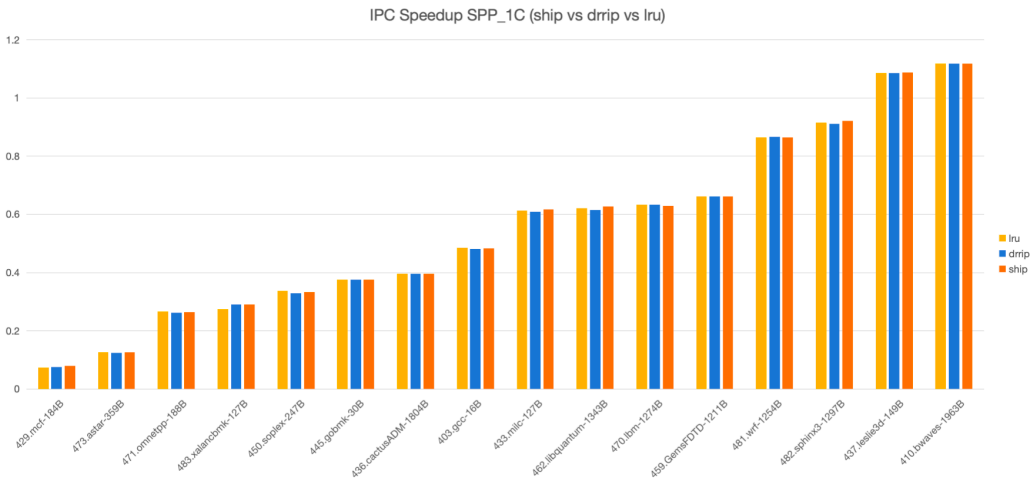

图 10. 单核不同替换算法下 SPP 的 IPC 数据

由图 11、12 也可以发现 Pythia 和 SPP 在 LLC 为 SHIP 替换策略下的性能更好，远超 LRU 和 DRRIP 替换策略，而且还能发现 Pythia 得到的增益比 SPP 的更大，多核情况下 LLC 才用 SHIP 替换策略和 Pythia 的组合带来了更好的性能表现。

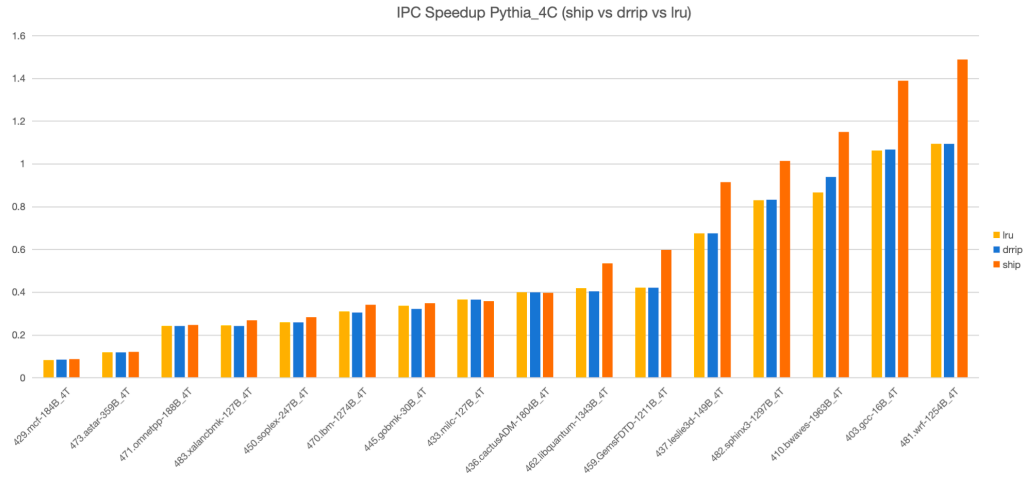

图 11. 图 11 四核不同替换算法下 Pythia 的 IPC 数据

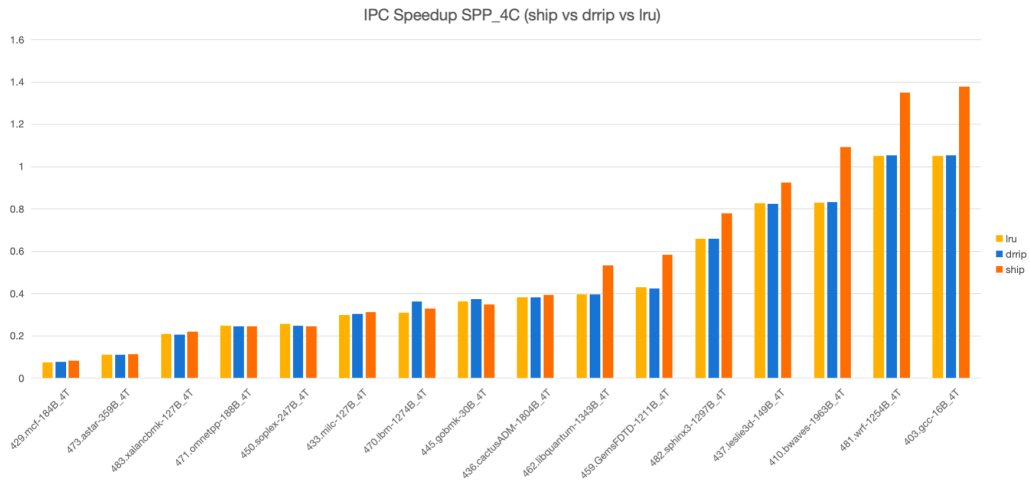

图 12. 四核不同替换算法下 SPP 的 IPC 数据

最后我们来看不同 MTPS 下各个预取器的表现情况，MTPS 用于表示 DRAM 的内存带宽性能，较大的 MTPS 说明 DRAM 带宽越大，在一个激进的预取测量下处理器的性能不会在较小的 MTPS 下带来显著的性能提升，从图 12 可以看到 Pythia 的表现是最好的，Bingo 因为其激进的预取策略在多个不同 MTPS 下都不占优，而 Pythia 因为考虑到了整体系统的带宽等元素（带宽也是 Pythia 的 feature 之一），能够在更低的 MTPS 下带来更好的性能表现，体现出了极强的自适应性。

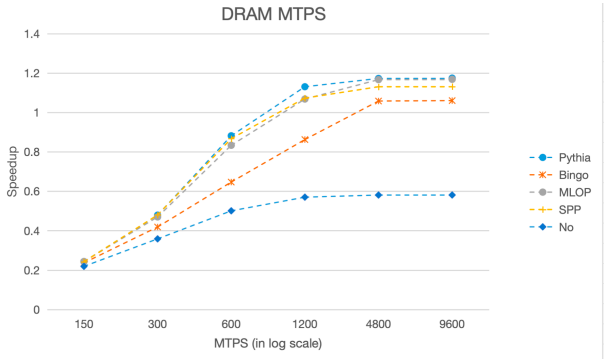

图 13. 不同 MTPS 下预取器的性能（410.bwaves-1963B）

6 总结与展望

Pythia 这篇论文是近些年发表的第一个基于强化学习的缓存硬件预取器，在性能上有显著的增强，同时也是体系结构设计与机器学习相结合的一个典范，体现了一种 Machine Learning for Computer Architecture 的领域交叉融合思想，在这篇论文之后有越来越多的基于机器学习算法的缓存预取策略或缓存替换策略被提出，同时也可以看到这篇论文不仅做了很多具有开创性的工作，还提供了一系列严谨的实验方法以及性能评估方法，可以进行参考研究。参考 Pythia 的思想，在未来我们可以更多的将 AI 与体系结构进行结合，这种跨领域的结合（特别是 AI）容易带来意外的发现，强化学习的思想也能够适用于多种体系结构相关问题上，更进一步能够提升体系结构的” 智能” 程度，在各种程序下处理器都能有良好的性能表现。

参考文献

[1] Rahul Bera, Konstantinos Kanellopoulos, Anant V. Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.